NASA-TM-108130

90

7N-61-TM

136176

P.15

ORIGINAL PAGE IS OF POOR QUALITY

GENERAL OBJECT-ORIENTED SOFTWARE DEVELOPMENT WITH ADA: A LIFE CYCLE APPROACH

CASE Technology Conference
April 1988

Ed Seidewitz
Code 554 / Flight Dynamics Analysis Branch

Goddard Space Flight Center
Greenbelt MD 20771
(301) 286-7631

Abstract

The effective use of Ada requires the adoption of modern software-engineering techniques such as object-oriented methodologies. A Goddard Space Flight Center Software Engineering Laboratory Ada pilot project has provided an opportunity for studying object-oriented design in Ada. The project involves the development of a simulation system in Ada in parallel with a similar FORTRAN development. As part of the project, the Ada development team trained and evaluated object-oriented and process-oriented design methodologies for Ada.

In *object-oriented* software engineering, the software developer attempts to model entities in the problem domain and how they interact. Most previous work on object-oriented methods has concentrated on using object-oriented ideas in software design and implementation. However, we have also found that object-oriented concepts can be used advantageously throughout the entire Ada software life-cycle. This paper provides a distillation of our experiences with object-oriented software development. It considers the use of entity-relationship and process/data-flow techniques for an object-oriented specification which leads smoothly into our design and implementation methods, as well as an object-oriented approach to reusability in Ada.

1. Introduction

Increased productivity and reliability from using Ada must come from innovative application of the non-traditional features of the language. However, past experience has shown that traditional development methodologies result in Ada systems that "look like a FORTRAN design" (see, for example, [Basili 85]). *Object-oriented* techniques provide an alternative approach to effective use of Ada. As the name indicates, the primary modules of an object-oriented design are *objects* rather than traditional functional procedures. Whereas a procedure models an action, an object models some *entity* in the problem domain, encapsulating both data about that entity and operations on that data. Ada is especially suited to this type of design because its package facility directly supports the construction of objects.

The Goddard Space Flight Center Software Engineering Laboratory is currently involved in an Ada pilot project to develop a system of about 60,000 lines (20,000 statements) [Nelson 86, McGarry 88]. This project has provided an opportunity to explore object-oriented software development methods for Ada. The pilot system, known as "GRODY", is an attitude dynamics simulator for the Gamma Ray Observatory (GRO) spacecraft and is based on the same requirements as a FORTRAN system being developed in parallel.

The GRODY team was initially trained both in the Ada language and in Ada-oriented design methodologies. The team specifically studied the methodology promoted by Grady Booch [Booch 83] and the PAMELA™ methodology of George Cherry [Cherry 85]. Following this, during a training exercise, the team also began synthesizing a more general approach to object-oriented design. At an early stage of the GRODY development effort, the team produced high-level designs for GRODY using each of these methodologies. Section 2 summarizes the comparison of methodologies made by the GRODY team.

PAMELA is a registered trademark of George W. Cherry.

(NASA-TM-108130) GENERAL OBJECT-ORIENTED SOFTWARE DEVELOPMENT WITH Ada: A LIFE CYCLE APPROACH (NASA) 15 p

N93-70958

Unclas

5207

29/61 0136176

PRECEDING PAGE BLANK NOT FILMED

Unfortunately, the system requirements given to our team were highly biased by past FORTRAN designs and implementations of similar systems. Therefore we began by recasting the requirements in a more language-independent way using the "Composite Specification Model" [Agresti 84, Agresti 87]. This method involves the use of state transition and entity-relationship techniques as well as more traditional data flow diagrams. We then designed the system to meet this specification, using object-oriented principles. The resulting design is, we believe, an improvement over the previous FORTRAN designs [Agresti 86]. The system is currently in final system testing.

Previous work by the present authors has concentrated on using object-oriented ideas in software design and implementation. This work resulted in a design method which synthesizes the best methods studied during the GRODY project [Seidewitz 86a, Seidewitz 86b]. However, we have found that object-oriented concepts can be used advantageously throughout the entire Ada software life-cycle [Stark 87]. Section 3 provides a distillation of our experience with GRODY and other Ada projects into an evolving life-cycle methodology.

2. Comparison of Methodologies

This section presents a comparison of design approaches to the GRO dynamics simulator, including the traditional functional approach used for the FORTRAN version, the Booch methodology, PAMELA and the general methodology developed by the team itself. It should be noted that the GRODY team was trained in the Booch and PAMELA methodologies in early 1985. Since then, both methodologies have evolved considerably, in many cases addressing in different ways the very issues that led us to develop our methodology. Nevertheless, as background motivation for the direction taken by the GRODY team, the comparison in this section is in terms of the 1985 versions of the methodologies.

2.1 Functional Design

The design of the FORTRAN version of the simulator is functionally-oriented. This design has a strong heritage in previous simulator and ground support systems. It consists of three major subsystems which interact as shown in figure 1. The "TRUTH MODEL" subsystem includes models of the spacecraft hardware, the external environment and the attitude dynamics; that is, the "real world" as opposed to the spacecraft control system. The SIMULATION CONTROL subsystem alternatively activates the SPACECRAFT CONTROL and TRUTH MODEL subsystems in a cyclic fashion. Each subsystem consists of a single *driver* subroutine which calls on a hierarchy of lower-level subroutines to perform the functions of the subsystem when activated by SIMULATION CONTROL. Data flow between subsystems, as well as system parameterization, is entirely though a set of global COMMON areas.

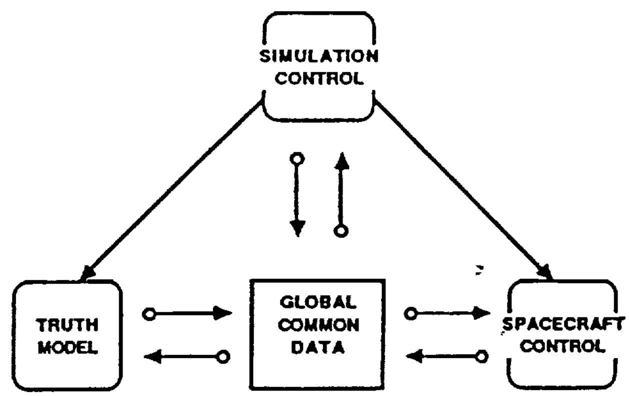

FIGURE 1 *FORTRAN Simulator Functional Design*

The strengths of this functional design lay in its relatively simple structure and direct implementation in FORTRAN. However, its main drawback is the complete lack of encapsulation of global data. The only restrictions on which code may access which global data are enforced by programmer discipline. This can lead, intentionally or not, to illicit corruption of global data by code in one part of the system which is unexpected by another part of the system. Further, most simulation parameters are hard-coded into the global common area, making the user interface for the system hard to modify and impossible to generalize.

2.2 Booch's Methodology

Grady Booch is, perhaps, the most influential advocate of object-oriented design in the Ada community [Booch 86b, Booch 87]. As learned by the GRODY team, Booch's methodology derives a design from a textual specification or informal design [Booch 83], an approach adopted from Abbott [Abbott 83]. The technique is to underline all the nouns and verbs in the specification. The objects in the design derive from the nouns; object operations derive from the verbs. Obviously, some judgment must be used to disregard irrelevant nouns and verbs and to translate the remaining concepts into design objects. Once the objects have been identified, the design can then be represented diagrammatically using a notation which shows the dependencies between Ada packages and tasks which implement the objects. Figure 2 shows such a diagrammatic top-level design for GRODY.

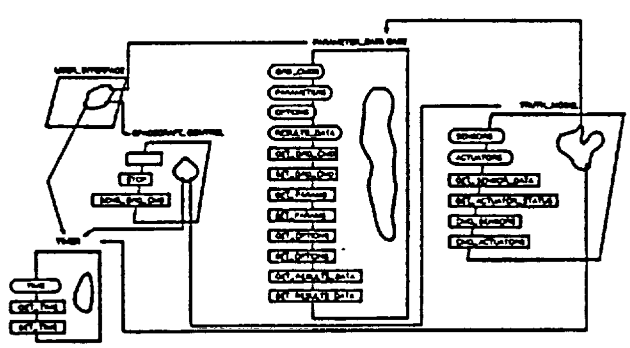

FIGURE 2 *Object-Oriented Simulator Design (Booch Methodology)*

The Booch design methodology contains all the basic framework of the object-oriented approach. However, application of this methodology to GRODY indicated that it was not readily applicable to sizable systems. The team found the graphical notation clear but not detailed or rigorous enough. Further, Booch gives no explicit method for diagramming a hierarchical decomposition of objects, which is needed for any sizable system. Booch's notation does not, therefore, seem to be a complete design notation. Note, however, that in more recent work Booch has extended the scope of the notation to address some of these shortcomings [Booch 87].

A second difficulty of Booch's methodology is in the technique for deriving the design from the specification text. This works well when the specification can be written concisely in a few paragraphs. However, when the system requirements are large, as with GRODY, this can be difficult. In addition, any attempt to use such a technique directly on a requirements document such as ours is doomed to failure due to the sheer size and complexity of the document. Realizing such drawbacks, Booch no longer advocates the use of this textual method, which was never actually intended for large systems development [Booch 86b]. Instead, he derives an object-oriented design from a data flow diagram based specification [Booch 86a, Booch 87]. However, from the published examples it is still unclear how to systematically apply this method to realistic systems.

2.3 PAMELA

The second methodology considered by the GRODY team was the Process Abstraction Method for Embedded Large Applications (PAMELA) developed by George Cherry [Cherry 85, Cherry 86]. PAMELA is oriented toward real-time and embedded systems. PAMELA is *process-oriented*, so a PAMELA design consists of a set of interacting *concurrent processes*. A well designed *process* is effectively a concurrent object, thus PAMELA is object-oriented in a general way.

PAMELA uses a powerful graphical notation without many of the drawbacks found in Booch's notation [Cherry 86]. During the PAMELA design processes, the designer successively decomposes processes into concurrent subprocesses until he reaches the level of primitive *single-thread* processes. The GRODY team found that PAMELA provides fairly explicit heuristics for constructing good processes. The designer uses these hints to construct the top-level processes from the system specification. The designer then recursively decomposes each non-primitive processes until only primitive processes remain. The primitive processes can then be coded as Ada tasks with a single thread of control. Non-primitive processes are simply packages of lower level processes and thus contain multiple threads of control. Figure 3 shows the top levels of a PAMELA design for GRODY.

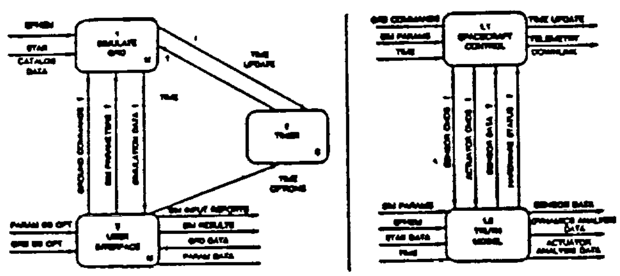

FIGURE 3 *PAMELA Simulator Design*

PAMELA's heuristics can be very effective when designing a real-time system that is heavily driven by external asynchronous actions. In other cases, however, they require considerable interpretation to be applicable. Although parts of GRODY might conceptually be concurrent (because GRODY simulates actions that happen in parallel in the real world), there is no requirement for concurrency in the simulation of these actions because GRODY does not have to interface with any active external entity (except the user). In addition, since GRODY runs on a sequential machine, the overhead of Ada tasking and rendezvous could greatly degrade the time performance of the system. Thus, one interpretation of PAMELA's principles might leave very large sections of GRODY as primitive single-thread processes, with only a few concurrent objects in the entire design. To proceed further in the decomposition, the designer has to rely more on intuition about what makes a good object and rely less on the methodology.

In fact, at the time that the GRODY team was using PAMELA, it provided no support for the decomposition and design of anything below the level of the primitive process, an Ada task [Cherry 85]. Since then, Cherry has added several concepts to the methodology, including the use of abstract data types [Cherry 86]. Recently he has introduced a major update of PAMELA known as "PAMELA 2" which is now explicitly object-oriented [Cherry 88]. In fact, PAMELA now stands for "Pictorial Ada Method for Every Large Application." It is still to early, however, to evaluate the generality of PAMELA 2 as an object-oriented methodology.

2.3 General Object-Oriented Development

As a result of the above experiences, the GRODY team developed its own object-oriented methodology which attempts to capture the best points of the object-oriented approaches studied by the team as well as traditional structured methodologies [Seidewitz 86a, Seidewitz 86b, Stark 87]. It is designed to be quite general, giving the designer the flexibility to explore design alternatives easily. It is also based on principles that guide the designer in constructing good object-oriented designs. This methodology was used to develop the complete detailed design for GRODY.

This general object-oriented development ("GOOD") methodology is based on general principles of abstraction, information hiding and design hierarchy discussed in the next section. These principles are less explicit than Booch's methodology or PAMELA, but they do provide a firm paradigm for generating and evaluating an object-oriented design. Indeed, as mentioned above, the team found the Booch and PAMELA design construction techniques restrictive, often necessitating the designer to rely on intuition for object-oriented design. The GOOD methodology is an attempt to codify this intuition into a basic set of principles that provide guidance while leaving the designer the flexibility to explore various design approaches.

In addition, we have also considered, independently of Booch, the transition from structured analysis [DeMarco 79] to object-oriented design in the context of the GOOD methodology, developing a technique known as *abstraction analysis* [Seidewitz 86a, Seidewitz 86b]. This technique is analogous to transform and transaction analysis used in structured design [Yourdon 78]. However, proceeding into object-oriented design from a structured analysis, by whatever means, requires an "extraction" of problem domain entities from traditional data flow diagrams. From an object-oriented viewpoint, it seems appropriate to instead *begin* a specification effort by identifying the entities in a problem domain and their interrelationships. Study is continuing on including such object-oriented system specification techniques in the GOOD methodology and on applying object-oriented principles throughout the Ada life cycle [Stark 87]. Section 3 will discuss this in more detail.

Figure 4 shows the actual design of the main part of GRODY. The *object diagram* notation [Seidewitz 86b] used in figure 4 shows the dependencies between the various objects which make up a system design, in a manner similar to Booch's diagrams. However, the object diagram notation also explicitly includes the idea of leveled composition of objects, like the PAMELA process graph notation. Moreover, as will be discussed in more detail in section 3, the designer may use object diagrams to express the design from the highest levels all the way down to the procedural level. (This capability has also been added to PAMELA 2 [Cherry 88].)

Since GRODY was derived from the same basic requirements as the FORTRAN design, there are similarities in the designs of the two systems. However, there are also some fundamental differences in the GRODY design that can be traced to the object-oriented methodology. For example, in GRODY the TRUTH MODEL is effectively passive, with the SPACECRAFT CONTROL calling on operations as needed to obtain sensor data and activate actuators. All sensor and command data is passed using these operations. This design approach was encouraged by viewing the TRUTH MODEL as an object with multiple operations rather than as a functional subsystem with a single driver.

The simulation timing of GRODY is also different from the FORTRAN design. The object-oriented methodology led to consideration of a "TIMER" object in GRODY which provides an abstraction of the simulation time. This utility object provides a common time reference for the SPACECRAFT CONTROL and TRUTH MODEL separate from the SIMULATION CONTROL loop. Unlike the FORTRAN design, in GRODY the "cycle times" of the SPACECRAFT CONTROL and TRUTH MODEL are not the same. The GRODY team chose to faithfully model, in the SPACECRAFT CONTROL abstraction, the timing of the actual spacecraft control software, which is not under user control. However, GRODY allows the simulation user to set the cycle time for the TRUTH MODEL over a fairly wide range, to allow the user to trade-off speed and accuracy as desired.

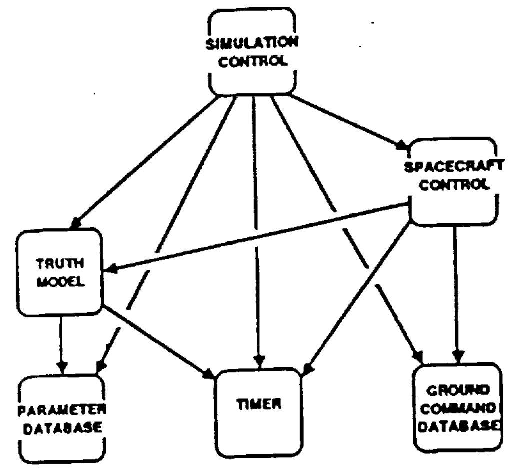

FIGURE 4 *Object-Oriented Simulator Design (GOOD Methodology)*

Finally, the PARAMETER DATABASE and GROUND COMMAND DATABASE objects encapsulate user settable parameters for the simulation. Similar data is contained in COMMON blocks in the FORTRAN design. This encapsulation of "global" data is typical of object-oriented designs. It provides both increased protection of the data encapsulated and increased opportunity for reuse. For example, the simulation parameters in the FORTRAN design are COMMON block parameters which must be hard-coded into the user interface code. (For simplicity the user interface modules have not been included in figure 4.) In the GRODY design, simulation parameters are identified by enumeration constants, which allows the user interface displays to be parameterized by external data files. This should greatly increase the reusability of the user interface.

The differences discussed above could probably have been incorporated into the FORTRAN design. However, it was largely the influence of the object-oriented approach which lead to their consideration for GRODY when they had not been considered in several previous designs of simulators for FORTRAN. Considerations of encapsulation and reusability indicate that the GRODY design may be "better" than the FORTRAN design. This is, of course, the goal of object-oriented methods. However, the true test of the merits of the GRODY design will only come from continuing studies of the comparative maintainability of the FORTRAN and Ada simulators.

In terms of the methodology itself, the team found the object diagram notation extremely useful for discussing the design during development. Further, the notation provided complete documentation of the design and was tailored specifically towards Ada. This made the transition to coding very smooth, and allowed the documentation to be readily updated as coding proceeded. By the end of coding, there were no major changes in the design and most changes that did occur were additions rather than alterations.

The object diagram notation evolved considerably during the GRODY project in response to continuing experience with its use. The lack of a specific methodology at the start of the GRODY project was a problem for the team, as was the continuing evolution of the methodology over the duration of the project. Further, the fact that managers were not familiar with the new methodology made the use of object diagrams difficult at reviews. Another problem was that the detail of the object diagrams and the emphasis on keeping the documentation up-to-date required a great deal of effort to maintain a rather large design notebook. The team clearly saw the great need for automated tools to support the methodology in this area. Consideration has also been given to extend the object diagram notation to better cover such topics as generics, abstract data types and large system components.

3. The GOOD Methodology

Section 2 described the background motivation of the GRODY team in developing the GOOD methodology and applying it to the full GRODY design. The experience with the Composite Specification Model and object-oriented design on GRODY, as well as experience on other Ada projects, has led to the continuing evolution of a comprehensive, integrated, object-oriented approach to software development, encompassing all phases of the software life cycle. This section provides an overview of the current GOOD life cycle approach.

3.1 Entities and Relationships

The modules of an object-oriented design are intended to primarily represent problem domain *entities*. From an object-oriented viewpoint, it seems appropriate to begin a software specification effort by identifying the entities in a problem domain and their interrelationships. Entity-relationships and data flow techniques can then complement each other, the former delineating the static structure problem domain and the latter defining the dynamic function of a system. This is similar to the "contextual" and "functional" views of the Composite Specification Model [Agresti 84, Agresti 87]. A close relation to the specification approach discussed here is described in some detail in [Bailin 88].

An *entity* is some individual item of interest in the problem domain. For example, consider the specification of GRODY. Several problem domain entities immediately come to mind: the spacecraft structure, sensors and thrusters on the spacecraft, the environment, etc. An entity is described in terms of the *relationships* into which it enters other objects. A spacecraft might be in a certain orientation, have certain thrusters, etc. Entities can also have *attributes*, such as spacecraft mass, which are data items describing the intrinsic properties of the entity.

To model the structure of the problem domain requires the identification of *entity types* which are groups of entities with the same types of attributes and relationships. For example, we may define a SPACECRAFT STRUCTURE entity type with SPACECRAFT MASS and DRAG COEFFICIENT attributes. All SPACECRAFT STRUCTURE entities have these attributes, but different individual entities have different specific *values* for the attributes.

A problem domain model must also include a specification of all possible relationships between various types of entities. These relationships may themselves have attributes and enter into other relationships. For example, the ATTITUDE STATE of a spacecraft describes its current orientation relative to inertial space and its current rotational motion. The ATTITUDE STATE is effectively a relationship between the spacecraft, the environment and the effect of any thruster firings used to reorient the spacecraft. This relationship has such attributes as the current spacecraft orientation and the spacecraft angular rotation rates.

The entity-relationship diagram (ERD) is a common graphical tool for entity-oriented specification [Chen 76]. Figure 5 shows an ERD for the GRODY problem domain. The notation for this diagram is based on [Ward 85]. Complex relationships such as ATTITUDE STATE are shown as *associative entities* on ERDs such as figure 5. Associative entities can be identified on an ERD by being connected to a relationship symbol by an arrow. Associative entities are "objectivizations" of relationships which may have attributes and enter into other relationships.

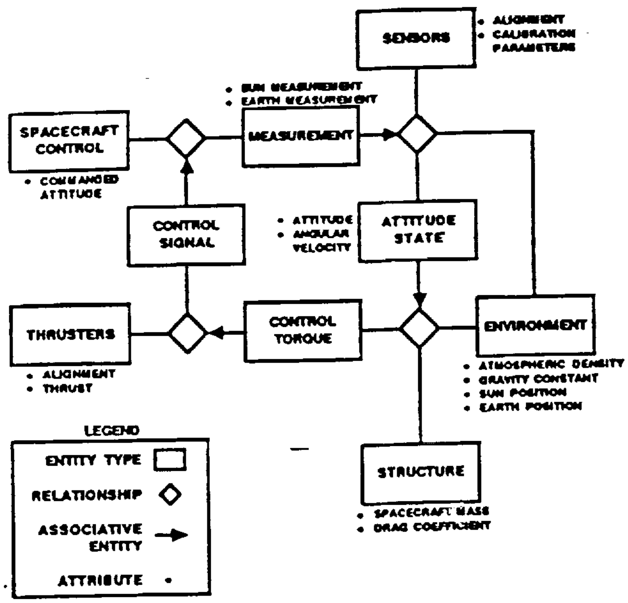

FIGURE 5 *Attitude Dynamics Entity-Relationship Diagram*

Figure 5 shows only a small part of the example problem domain. It would grow as additional entities and relationships are added to describe additional parts of the problem domain. As the specification grows, a complete ERD can quickly become cumbersome. It is possible to "level" ERDS showing complex entities on high-level diagrams which enter into *composite* relationships. These are then broken down in lower-level diagrams. An extended *data dictionary* notation is also useful as a textual representation of entity type and relationship definitions. In addition, the data dictionary provides a common basis for data definition between the static and the dynamic views of the problem domain.

3.2 Processes and Data Flow

ERDs show all *possible* relationships between different types of entities. They do not show the *actual* relationships between specific entities at specific points in time, nor how these actual relationships change over time. Data flow techniques, however, provide exactly this dynamic view. Traditional data flow diagrams (DFDs) show the flow of data between functional transformations. We will, instead, diagram the flow of data between *processes* which represent the dynamic view of one or more entities in the problem domain. A process is effectively a state machine which accepts input stimuli, reacts to it and produces output stimuli, possibly modifying some internal state data. It has no "operations" as such, only stimuli and responses. These *stimuli* may be either in the form of *data flow* or pure control *signals*.

To construct a dynamics data flow model, one needs to identify those *active entities* which have associated processes. For each relationship in the static entity-relationship model, we choose one of the related entity types to be active. This entity type has an associated process which is charged with maintaining the state of the relationship in response to internal and external stimuli. Note that an entity type may be active relative to one relationship and passive relative to another, and that associative entities may be active or passive.

For example, consider a simplified attitude dynamics simulation system similar to GRODY. The *attitude* of a spacecraft is its orientation relative to inertial space, and an attitude dynamics simulator models the *rotational* motion of the spacecraft in response to external disturbances and the spacecraft control system. Figure 5 describes the problem domain for such a system. The active entities in this case interact in a control loop outlined in figure 6. All the processes shown on figure 6 are associated with active entities on figure 5. A data item flowing on a diagram such as figure 6 may be a passive entity, an attribute or any other composite data item or data element defined in the data dictionary.

The dynamic model must also provide a specification for each individual process. This specification should include a textual description of the object as well as a listing of all inputs and outputs. The process specification also provides a place to include "non-functional" requirements such as timing and accuracy constraints. However, the main point of a process specification is to detail the function of the process. This can be in the form of structured English, a state transition diagram or some other appropriate notation, such as differential equations for the time evolution of the spacecraft attitude.

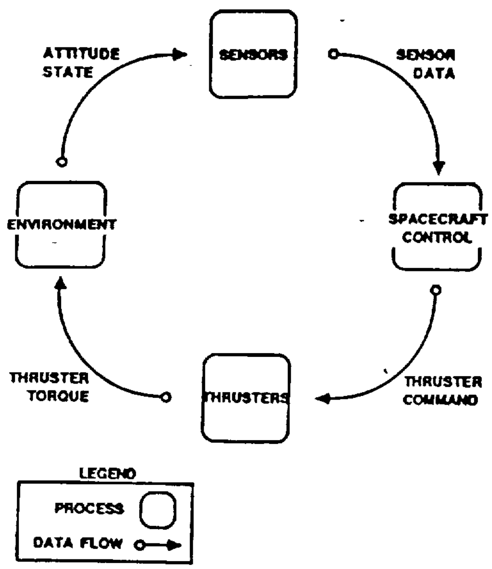

FIGURE 6 *Attitude Dynamics Data Flow Diagram*

The function of a process can also be given by a lower-level data flow diagram. Decomposition can continue recursively on all diagrams until all processes have been decomposed into primitive functions and states. This results in a leveling similar to the leveling of traditional DFDs. However, unlike DFDs, each object at each level of a process-data-flow diagram specification has a complete process specification. Each process must also be associated a reasonable problem domain entity independently of its decomposition.

3.3 Object Identification

The intent of an object is to represent a problem domain entity and any associated process. The concept of *abstraction* deals with how an object presents this representation to other objects [Booch 86b, Dijkstra 68]. Ideally, the objects in a design should directly reflect the problem domain entities identified during system specification. However, various design considerations may require splitting or grouping of objects and there will almost always be additional objects in a design to handle "executive" and "utility" functions. Thus there is a spectrum of levels of abstraction of objects in a design, from objects which closely model problem domain entities to objects which really have no reason for existence [Seidewitz 86b]. The following are some points in this scale, from strongest to weakest:

Entity Abstraction - An object represents a useful model of a problem domain entity or class of entities.

Action Abstraction - An object provides a generalized set of operations which all perform similar or related functions (this is similar to the idea of a "utility" object in [Booch 87]).

Subsystem Abstraction - An object groups together a set of objects and operations which are all related to a specific part of a larger system (this is similar to the "subsystem" concept in [Booch 87]).

The stronger the abstraction of an object, the more details are suppressed by the abstract concept. The principle of *information hiding* states that such details should be kept secret from other objects [Booch 87, Parnas 79], so as to better preserve the abstraction modeled by the object.

3.4 Design Hierarchies

The principles of abstraction and information hiding provide the main guides for creating "good" objects. These objects must then be connected together to form an object-oriented design. This design is represented using the graphical object diagram notation [Seidewitz 86b].

The construction of an object-diagram-based design is mediated by consideration of two orthogonal hierarchies in software system designs [Rajlich 85]. The *composition* hierarchy deals with the composition of larger objects from smaller component objects. The *seniority* hierarchy deals with the organization of a set of objects into "layers". Each layer defines a *virtual language extension* which provides services to senior layers [Dijkstra 68]. A major strength of object diagrams is that they can distinctly represent these hierarchies.

The composition hierarchy is directly expressed by *leveling* object diagrams (see figure 7). At its top level, any complete system may be represented by a single object which interacts with *external objects*. Beginning at this system level, each object can then be refined into component objects on a lower-level object diagram, designed to meet the specification for the object. The result is a leveled set of object diagrams which completely describe the structure of a system. At the lowest level, objects are completely decomposed into *primitive objects* such as procedures, tasks and internal state data stores. At higher levels, object diagram leveling can be used in a manner similar to Booch's "subsystems" [Booch 87].

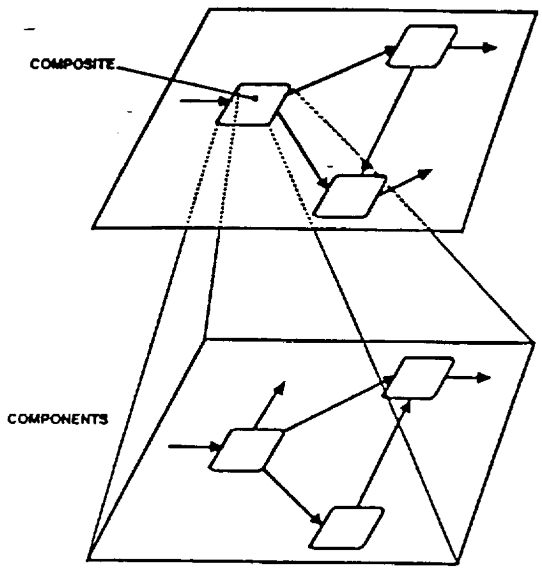

FIGURE 7 *Composition Hierarchy*

The seniority hierarchy is expressed by the topology of connections on a single object diagram (see figure 8). An arrow between objects indicates that one object calls *one or more* of the operations provided by another object. Any layer in a seniority hierarchy can call on any operation in junior layers, but *never* any operation in a senior layer. Thus, all cyclic relationships between objects must be contained within a virtual language layer. Object diagrams are drawn with the seniority hierarchy shown vertically. Each senior object can be designed as if the operations provided by junior layers were "primitive operations" in an extended language. Each virtual language layer will generally contain several objects, each designed according to the principles of abstraction and information hiding.

3.5 System Design

The main advantage of a seniority hierarchy is that it reduces the coupling between objects. This is because all objects in one virtual language layer need to know nothing about senior layers. Further, the centralization of the procedural and data flow control in senior objects can make a system easier to understand and modify.

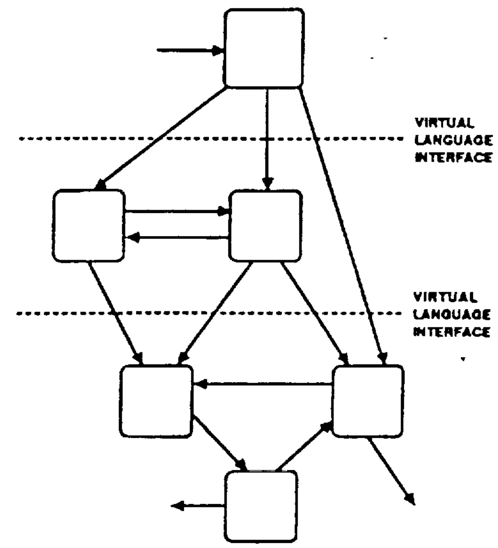

FIGURE 8 *Seniority Hierarchy*

However, this very centralization can cause a messy bottleneck. In such cases, the complexity of senior levels can be traded off against the coupling of junior levels. The important point is that the strength of the seniority hierarchy in a design can be chosen from a *spectrum* of possibilities, with the best design generally lying between the extremes. This gives the designer great power and flexibility in adapting system designs to specific applications.

Figure 9 shows one possible preliminary design for the ATTITUDE SIMULATOR. For simplicity, the sensors and thrusters are represented by a single "SPACECRAFT HARDWARE" object in figure 9. Note that, by convention, the arrow labeled "RUN" is the initial invocation of the entire system. In preliminary design diagrams such as figure 4, it is sometimes convenient to show what data flows along certain control arrows, much in the manner of structure charts [Yourdon 78] or "Buhr charts" [Buhr 84]. These annotations will not appear on the final object diagrams.

In figure 9, the junior level components do not interact directly. All data flow between junior level objects must pass through the senior object, though each object still receives and produces all necessary data (for simplicity not all data flow is shown in figure 9). This design is somewhat like an object-oriented version of the structured designs of Yourdon and Constantine [Yourdon 78].

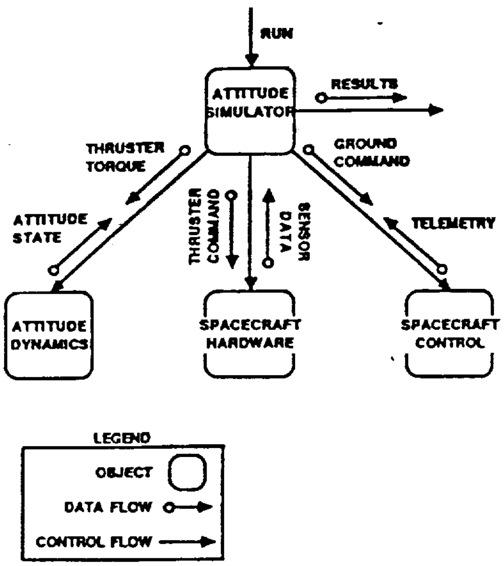

FIGURE 9 *Centralized Design*

We can remove the data flow control from the senior object and let the junior objects pass data directly between themselves, using operations within the virtual language layer (see figure 10). The senior object has been reduced to simply activating various operations in the virtual machine layer, with very little data flow.

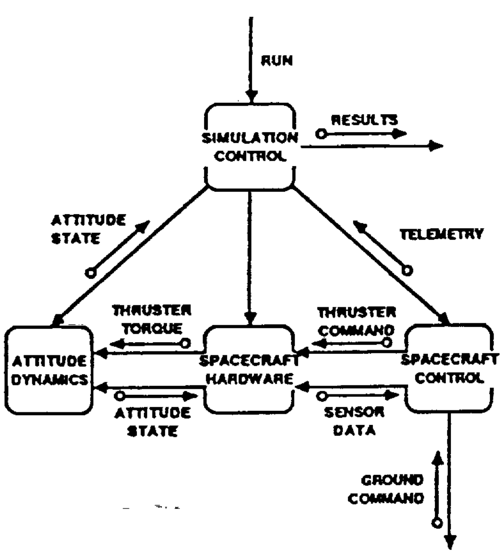

FIGURE 10 *Design with Decentralized Data Flow*

We can even remove the senior object completely by distributing control among the junior level objects (see figure 11). The splitting of the RUN control arrow in figure 11 means that the three objects are activated *simultaneously* and that they run *concurrently*. The seniority hierarchy has collapsed, leaving a ***homologous*** or non-hierarchical design [Yourdon 78] (no *seniority* hierarchy, that is; the composition hierarchy still remains).

A design which is decentralized like figure 11 at all composition levels is very similar to what would be produced by the PAMELA methodology [Cherry 86]. In fact, it should be possible to apply PAMELA design criteria to the upper levels of an object diagram based design of a highly concurrent system. All concurrent objects would then be composed, at a certain level, of objects representing certain process "idioms" [Cherry 86]. Below this level concurrency would generally no longer be advantageous.

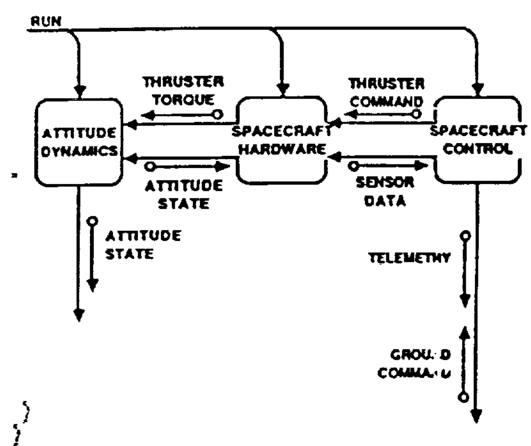

FIGURE 11 *Decentralized Design*

To complete the design, we need to add a virtual language layer of utility objects which preserve the level of abstraction of the problem domain entities. In the case of the ATTITUDE SIMULATOR these objects might include VECTOR, MATRIX, GROUND COMMAND and simulation PARAMETER types. Figure 12 shows how these objects might be added to the simulator design of Figure 10.

Figure 12 gives one complete level of the design of the ATTITUDE SIMULATOR. Note that figure 12 does not include the data flow arrows used in earlier figures. When there are several control paths on a complicated object diagram, it rapidly becomes cumbersome to show data flows. Instead, *object descriptions* for each object on a diagram provide details of the data flow.

An object description includes a list of all operations provided by an object and, for each arrow leaving the object, a list of operations used from another object. We can identify the operations provided and used by each object in terms of the specified data flow and the designed control flow. The object description can be produced by matching data flows to operations. For example, the description for the ATTITUDE DYNAMICS object in figure 12 might be:

```
Provides:
 procedure Initialize;
 procedure Integrate (For_Duration: in DURATION);
 procedure Apply (Torque: In VECTOR);
 function Current_Attitude return ATTITUDE;
 function Current_Angular_Velocity
  return VECTOR;

Uses:
 5.0 LINEAR ALGEBRA
   Add (Vector)
   Dot
   Multiply (Scalar)
   Multiply (Matrix)

 6.0 PARAMETER DATABASE
   Get
```

We could next proceed to refine the objects used in figure 12 and recursively construct lower level object diagrams. These lower level designs must meet the functionality of the system specification and provide the operations listed in the object description. The design process continues recursively until the entire system is designed and all objects are completely decomposed.

The GRODY design of figure 4 is basically the same as the example design of figure 12. However, the GRODY team chose to simplify the design by combining the ATTITUDE DYNAMICS and SPACECRAFT HARDWARE objects into a single TRUTH MODEL *subsystem object*, similar to the corresponding subsystem in the FORTRAN design. Further, in GRODY, the LINEAR ALGEBRA functions are part of a UTILITIES module not shown in figure 4.

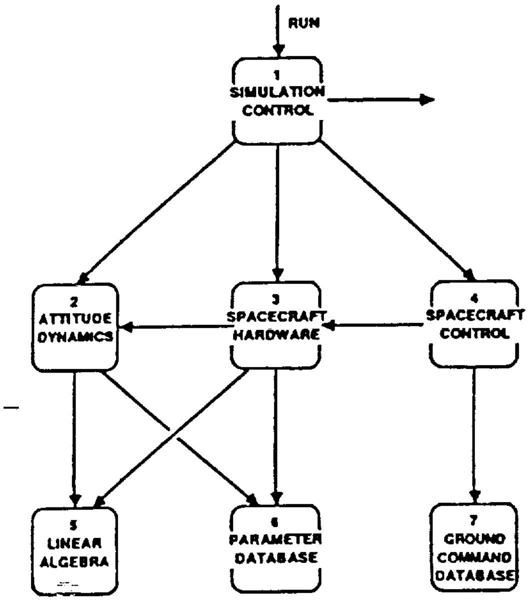

FIGURE 12 *Attitude Dynamics Simulator Design*

3.5 Implementation

The transition from an object diagram to Ada is straightforward. Package specifications are derived from the list of operations provided by an object. For the ATTITUDE DYNAMICS object the package specification is:

```
package Attitude_Dynamics is

  subtype ATTITUDE is Linear_Algebra.MATRIX;

  procedure Initialize;
  procedure Integrate
    ( For_Duration : in DURATION );
  procedure Apply
    ( Torque : in Linear_Algebra.VECTOR );

  function Current_Attitude
    return ATTITUDE;
  function Current_Angular_Velocity
    return Linear_Algebra.VECTOR;

end Dynamics;
```

The package specifications derived from the top level object diagram can either be made library units or placed in the declarative part of the top level Ada procedure. For lower level object diagrams the mapping is similar, with component package specifications being nested in the package body of the composite object. States are mapped into package body variables. This direct mapping produces a highly nested program structure. Alternatively, some or all of these packages can be made library units or even reused from an existing library. However, this may require additional packages to contain data types and state variables used by two or more library units. Nevertheless, experience has shown that, to promote reusability and reduce the compilation burden, it is best to avoid nesting of code [Godfrey 87], though it is important to retain leveling in the design.

The process of transforming object diagrams to Ada is followed down all the object diagram levels until we reach the level of implementing individual subprograms. Low-level subprograms can be designed and implemented using traditional functional techniques. They should generally be coded as subunits, rather than being embedded in package bodies.

The clear definition of abstract interfaces in an object-oriented design can also greatly simplify testing. When testing an object, there is a well defined "virtual language" of operations it requires from objects at a junior level of abstraction, some of which may be stubbed-out for initial testing. Further, object-oriented composition encourages incremental integration testing, since the "unit testing" of a composite object really consists of "integration testing" the component objects at a lower level of abstraction.

3.7 Reuse

The concept of *generic* objects provides a powerful tool for reusability. Generic parameters may be used to cut the dependencies of a general object on other specific objects, allowing the general object to be reused in similar but different contexts. Consider, for example, a general numeric integrator with the following package specification:

```
generic

  type REAL is digits <>;
  type STATE_VECTOR is
    array (INTEGER range <>) of REAL;
  with function State_Derivative
    ( T : DURATION; -- from reference time
      X : STATE_VECTOR )
    return STATE_VECTOR;

package Generic_Integrator is

  procedure Integrate
    ( For_Duration : in DURATION );
  function Current_State
    return STATE_VECTOR;
  procedure Initialize ....;

end Generic_Integrator;
```

This package provides the ability to numerically integrate a vector differential equation with an arbitrary state vector size. The "Integrate" procedure can be implemented as a vector equation, or as a set of individual real-valued functions. To implement it as a single vector equation we will need the operations provided by a LINEAR ALGEBRA object. These operations can be incorporated in two ways. One possibility is to make the operations needed into generic formal parameters. Another is to have the body of the integrator depend directly on LINEAR ALGEBRA.

Each of these methods has advantages and drawbacks. Using generic formal subprograms enhances reusability by making the component self-contained, but if too many are needed the interface becomes complex. Depending on LINEAR ALGEBRA within the GENERIC INTEGRATOR makes a cleaner interface, but couples the generic package to another library unit. The GRODY team has used both methods. Figure 13 shows the composition of GENERIC INTEGRATOR assuming the latter choice.

Figure 14 shows a use of the GENERIC INTEGRATOR in the composition of the ATTITUDE DYNAMICS object. The component object ATTITUDE INTEGRATOR is an instantiation of the GENERIC INTEGRATOR object. The generic object is instantiated in figure 14 with the ATTITUDE EQUATION subprogram as the generic actual parameter. Most of the ATTITUDE DYNAMICS operations are shown in figure 14 as component procedures, represented by rectangles. The "Integrate" operation, however, is directly *inherited* from the ATTITUDE INTEGRATOR object.

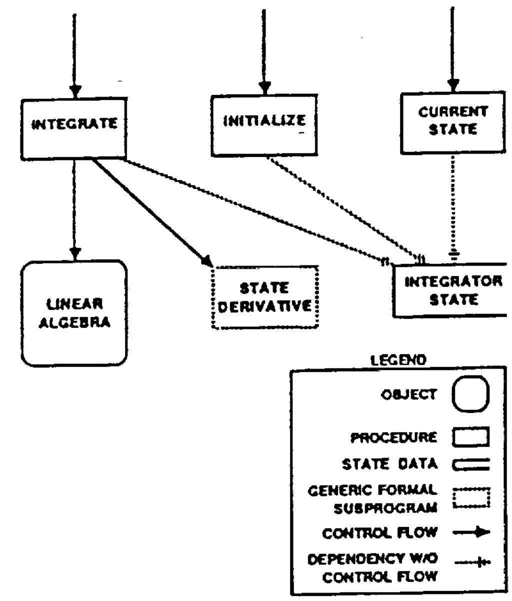

Figure 13 *Generic Integrator Object Composition*

Ada features such as generic packages are useful *tools*, but language features are not sufficient to guarantee high levels of software reuse. What is also needed is an *approach* to specifying and designing reusable components. Using an object-oriented approach is useful not because object-oriented design is essential for reuse, but because the underlying concepts are. These crucial concepts are abstraction, information hiding, levels of virtual languages (often called virtual "machines") and inheritance [Parnas 79, Cox 86].

Smalltalk's subclassing [Goldberg 83] provides an elegant means of supporting inheritance. Ada does not directly support inheritance, but the concept can be simulated by using "call-throughs." A call-through is a subprogram that has little function other than to call on another package's subprogram. To simulate inheritance when implementing the Attitude_Dynamics package the subprogram Integrate would be respecified in the Attitude_Dynamics package, with the subprogram body in Attitude_Dynamics calling on the corresponding operation from Attitude_Integrator.

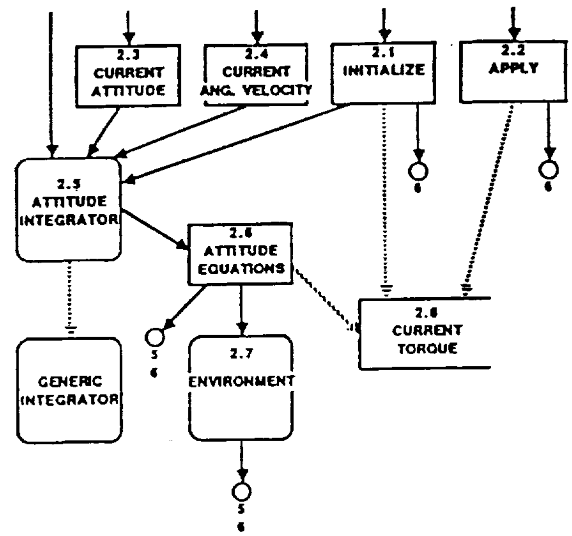

FIGURE 14 *Attitude Dynamics Object Composition*

This technique is clearly less elegant than Smalltalk subclassing, but it also has advantages. First, Ada allows inheritance from more than one object. Second, Smalltalk forces the inheritance of *all* operations and data. An operation can be overridden, but not removed, from a class. The Ada specification of the composite package gives the developer precise control over which operations and data items are visible or accessible. (See [Seidewitz 87] for a more detailed discussion of Ada and the concept of inheritance.)

4. Conclusion

The GRODY project has provided an extremely valuable experience in the application of object-oriented principles to a real system. This experience guided the creation of the GOOD methodology which is now being used on an increasing number of projects inside and outside of the Goddard Space Flight Center. As with any pilot project, some of the major products of GRODY are the lessons learned along the way.

As part of the GRODY project, a detailed assessment has been made of the team's experiences during design [Godfrey 87]. At this time, however, most of the observations must remain qualitative. Nevertheless, it is clear that the GRODY design is significantly different from previous FORTRAN simulator designs [Agresti 86].

It also became clear during the GRODY project that the GOOD methodology does not fit comfortably into the traditional life cycle management model. At the very least, the design phase should be extended and design reviews should occur at different points in the life cycle. The preliminary design review should occur later in the design phase and should include detailed object diagrams for the upper levels of the system, perhaps down to the level at which the design becomes more procedural than object-oriented. The critical design review would then include the detailed procedural designs, perhaps using an Ada-based design language. This review might actually take place as a series of incremental reviews of different portions of the design. This later approach is supported by the well-defined modularity of an object-oriented design.

The traditional functional viewpoint provides a comprehensive framework for the entire software life cycle. This viewpoint reflects the action-oriented nature of the machines on which software is run. The object-oriented approach discussed here can also provide a comprehensive view of the life cycle. The object-oriented viewpoint, however, reflects the natural structure of the problem domain rather than the implicit structure of our hardware. Thus, it provides a "higher-level" approach to software development which decreases the distance between problem domain and software solution. By making complex software easier to understand, this simplifies both system development and maintenance.

References

[Abbott 83]
Abbott, R. J. "Program Design by Informal English Description," *Communications of the ACM*, September 1983.

[Agresti 84]
Agresti, William W. "An Approach to Developing Specification Measures," *Proceedings of the 9th Annual Software Engineering Workshop*, GSFC Document SEL-84-004, November 1984.

[Agresti 86]
Agresti, William W., et. al. "Designing with Ada for Satellite Simulation: a Case Study," *Proceedings of the 1st International Conference on Ada Applications for the Space Station*, June 1986.

[Agresti 87]
Agresti, William W. *Guidelines for Applying the Composite Specification Model (CSM)*, GSFC Document SEL-87-003, June 1987.

[Bailin 88]
Bailin, Sidney C. and J. Michael Moore. "An Object-Oriented Specification Method for Ada," to be presented at the *Fifth Washington Ada Symposium*, June 1988.

[Basili 85]
Basili, V. R., et. al. "Characterization of an Ada Software Development," *Computer*, September 1985.

[Booch 83]
Booch, Grady. *Software Engineering with Ada*, Benjamin/Cummings, 1983.

[Booch 86a]
Booch, Grady. "Object-Oriented Software Development," *IEEE Transactions on Software Engineering*, February 1986.

[Booch 86b]
Booch, Grady. *Software Engineering with Ada. 2nd Edition*, Benjamin/Cummings, 1986.

[Booch 87]
Booch, Grady. *Software Components with Ada*, Benjamin/Cummings, 1987.

[Buhr 84]
Buhr, R. J. A. *System Design with Ada*, Prentice-Hall, 1984.

[Chen 76]
Chen, P. "The Entity-Relationship Model -- Toward a Unified View of Data," *ACM Transactions on Data Base Systems*, March 1976.

[Cherry 85]
Cherry, George W. PAMELA Course Notes, Thought**Tools, 1985.

[Cherry 86]
Cherry, George W. *PAMELA Designer's Handbook*, Thought**Tools, 1986.

[Cherry 88]
Cherry, George W. *PAMELA 2: An Ada-Based Object-Oriented Design Method*, Thought**Tools, February 1988.

[Cox 86]
Cox, Brad. *Object-Oriented Programming: an Evolutionary Approach*, Addison-Wesley, 1986.

[Dijkstra 68]
Dijkstra, Edsgar W. "The Structure of the 'THE' Multiprogramming System," *Communications of the ACM*, May 1968.

[Godfrey 87]
Godfrey, Sara, Carolyn Brophy, et. al. *Assessing the Ada Design Process and its Implications: a Case Study*, GSFC Document SEL-87-004, July 1987.

[Goldberg 83]
Goldberg, Adele and David Robson. *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1983.

[McGarry 88]
McGarry, Frank and William Agresti. "Measuring Ada for Software Development in the Software Engineering Laboratory (SEL)," *Proceedings of the 21st Hawaii International Conference on Software Engineering*, January 1988.

[Nelson 86]
Nelson, Robert W. "NASA Ada Experiment -- Attitude Dynamic Simulator," *Proceedings of the Washington Ada Symposium, March 1986.*

[Parnas 72]
Parnas, David L. "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, December, 1972.

[Parnas 79]
Parnas, David L. "Designing Software for Ease of Expansion and Contraction," *IEEE Transactions on Software Engineering*, March 1979.

[Rajlich 85]
Rajlich, Vaclav. "Paradigms for Design and Implementation in Ada," *Communications of the ACM*, July 1985.

[Seidewitz 86a]
Seidewitz, Ed and Mike Stark. "Towards a General Object-Oriented Software Development Methodology," *Proceedings of the 1st International Conference on Ada Applications for the Space Station*, June 1986.

[Seidewitz 86b]
Seidewitz, Ed and Mike Stark. *General Object-Oriented Software Development*, GSFC Document SEL-86-002, August 1986.

[Seidewitz 87]
Seidewitz, Ed. "Object-Oriented Programming in Smalltalk and Ada", *Proceedings of the Conference on Object-Oriented Programming, Languages, Systems and Applications*, October 1987.

[Stark 87]
Stark, Mike and Ed Seidewitz. "Towards a General Object-Oriented Ada Lifecycle," *Proceedings of the Joint Conference on Ada Technology / Washington Ada Symposium*, March 1986.

[Ward 85]
Ward, Paul T. and Stephen J. Mellor, *Structured Development for Real-Time Systems*, Yourdon Press, 1985.

[Yourdon 78]
Yourdon, Edward and Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Yourdon Press, 1978.